IT 1204
Section 2.0

Data Representation and Arithmetic

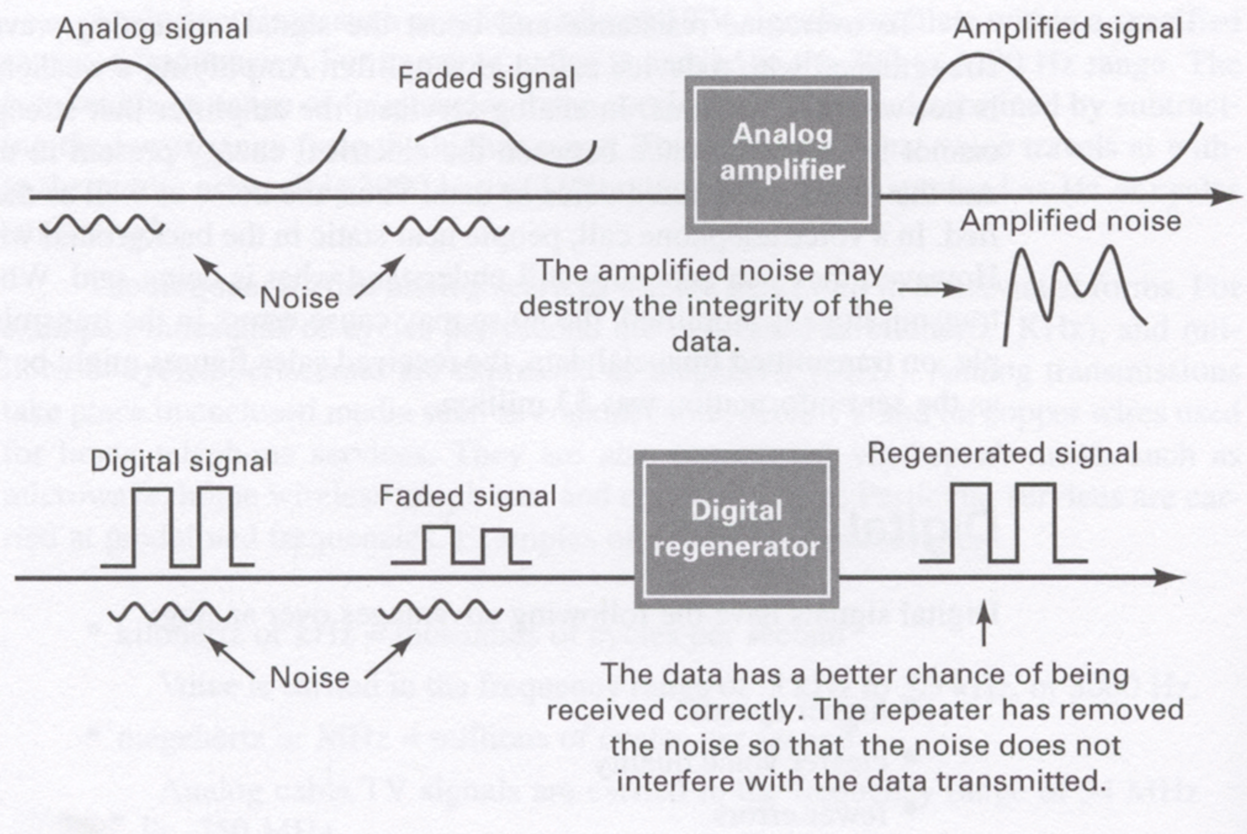
What is Analog and Digital

- The interpretation of an analog signal would correspond to a signal whose key characteristic would be a continuous signal
- A digital signal is one whose key characteristic (e.g. voltage or current) fall into discrete ranges of values
- Most digital systems utilize two voltage levels

UCSC

BIT

Advantage of Digital over Analog

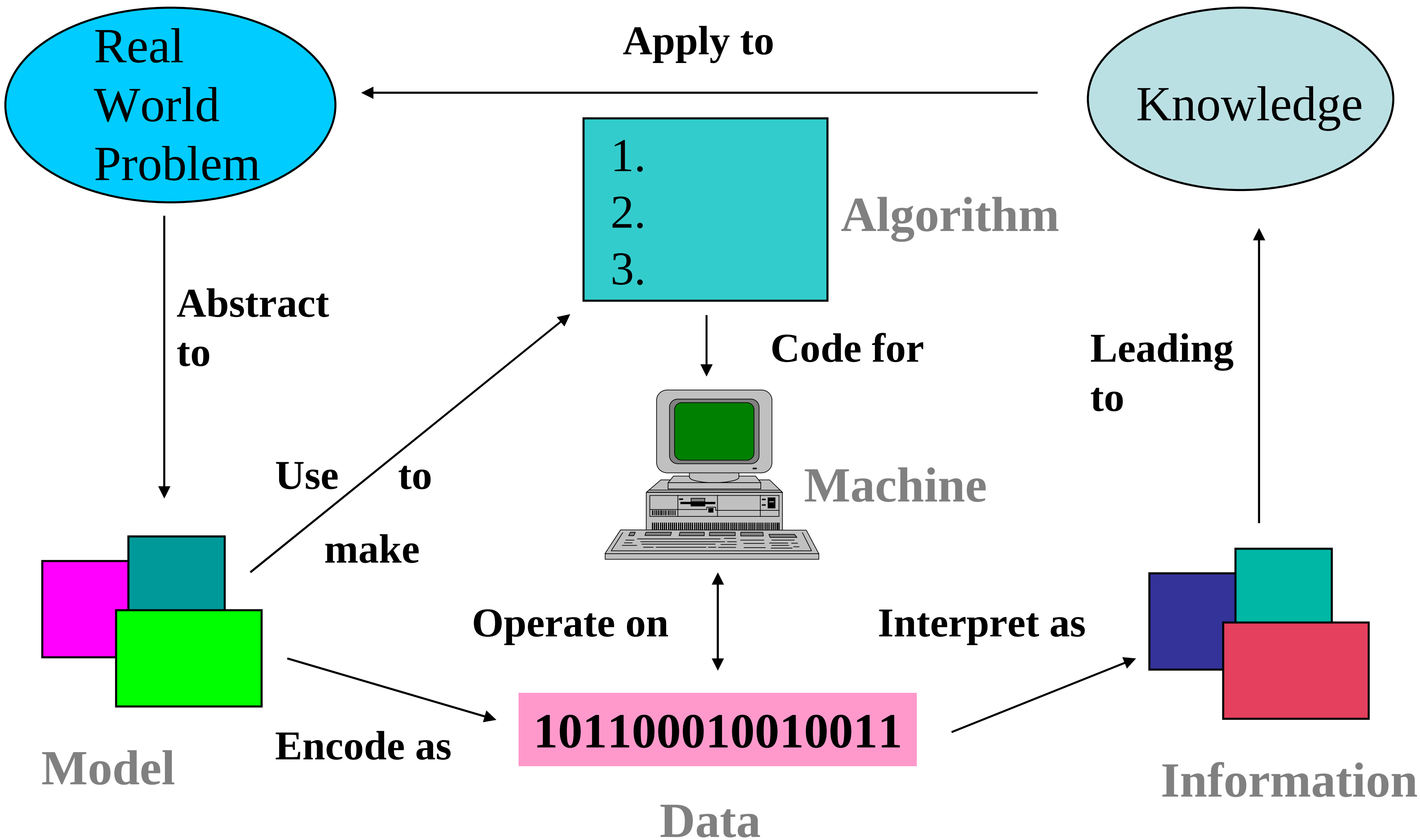


Noise amplified on analog lines; eliminated on digital service.

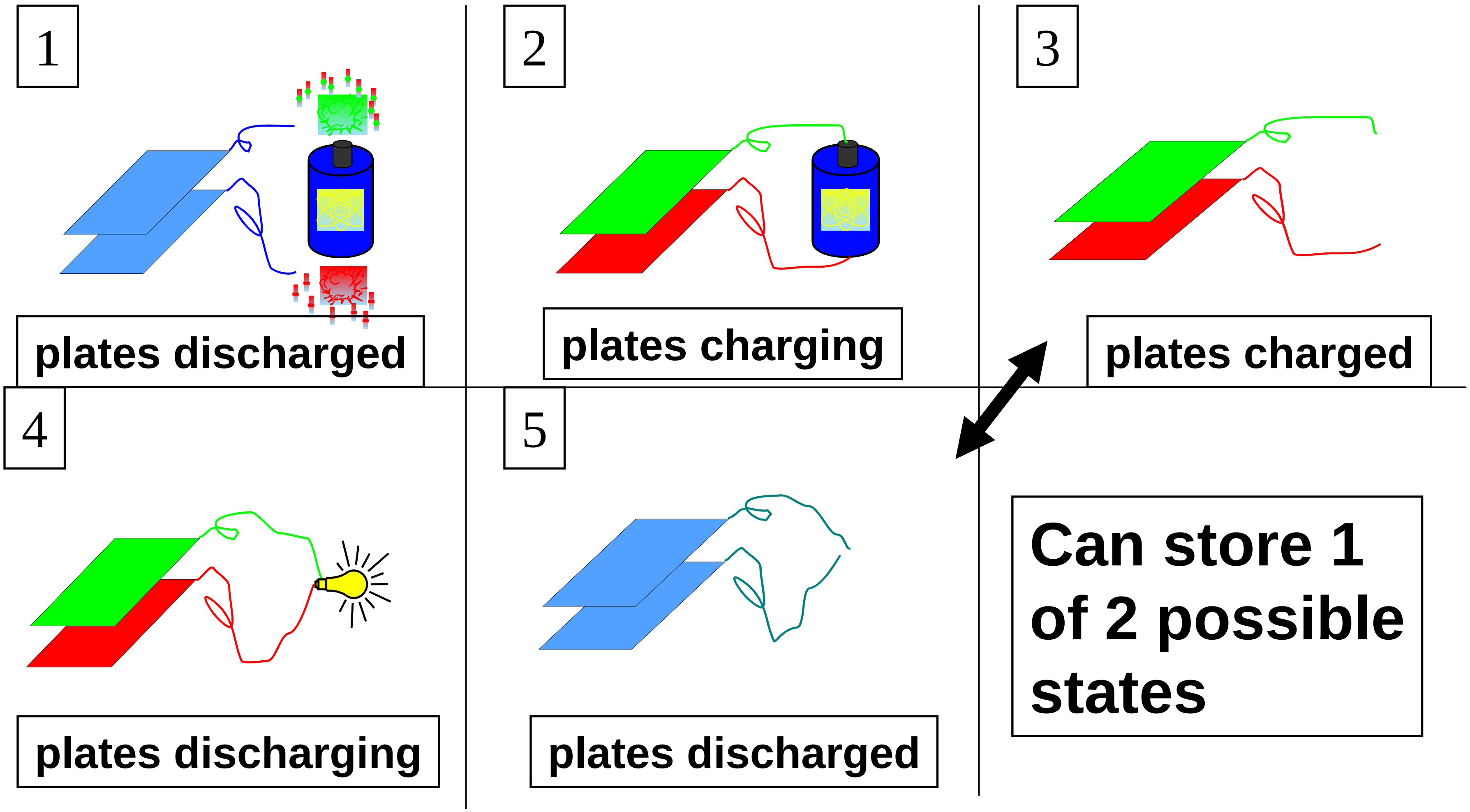
What is a bit

- A **bit** is a binary digit, the smallest increment of data on a machine. A **bit** can hold only one of two values: **0** or **1**
- Because **bits** are so small, you rarely work with information one **bit** at a time.

What is a bit


Real World Problem
Apply to
Knowledge
1.
2.
3.
Algorithm
Abstract to
Code for
Leading to
Machine
Use to make
Operate on
Interpret as
Model
Encode as
101100010010011
Data
Information

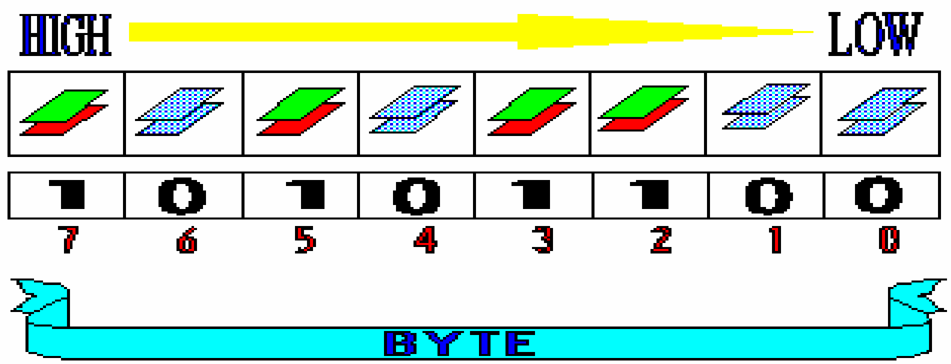
Bit Storage - Capacitor



Can store 1 of 2 possible states

What is a bit

- **Byte** is an abbreviation for "binary term". A single byte is composed of 8 consecutive bits capable of storing a single character



Storage Hierarchy

- 8 Bits = 1 Byte
- 1024 Bytes = 1 Kilobyte (KB)
- 1024 KB = 1 Megabyte (MB)
- 1024 MB = 1 Gigabyte (GB)
- A **word** is the default data size for a processor

Numbering System

- Decimal System
  - Alphabet = { 0,1,2,3,4,5,6,7,8,9 }
- Octal System
  - Alphabet = { 0,1,2,3,4,5,6,7 }
- Hexadecimal System
  - Alphabet = { 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F }
- Binary System
  - Alphabet = { 0,1 }

Converting decimal to binary

123			**1111011**
÷2			
61	→	remainder **1**	↑
÷2			
30	→	remainder **1**	
÷2			
15	→	remainder **0**	
÷2			
7	→	remainder **1**	
÷2			
3	→	remainder **1**	
÷2			
1	→	remainder **1**	
÷2			
0	→	remainder **1**	

Converting decimal to binary

123			**1111011**
÷2			
61	→	remainder **1**	
÷2			
30	→	remainder **1**	
÷2			
15	→	remainder **0**	
÷2			
7	→	remainder **1**	
÷2			
3	→	remainder **1**	
÷2			
1	→	remainder **1**	
÷2			
0	→	remainder **1**	

Most significant bit

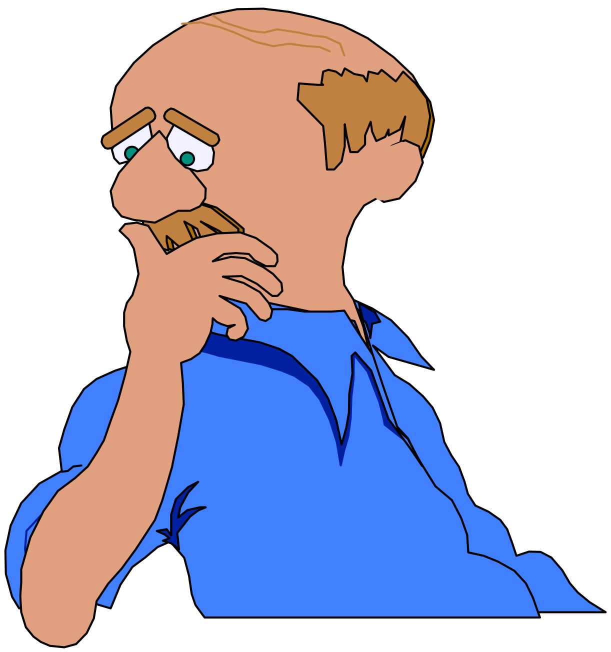
Converting decimal to binary

123			**1111011**
÷2			
61	→	remainder **1**	
÷2			
30	→	remainder **1**	
÷2			
15	→	remainder **0**	
÷2			
7	→	remainder **1**	
÷2			
3	→	remainder **1**	
÷2			
1	→	remainder **1**	
÷2			
0	→	remainder **1**	

Least significant bit

Your turn

Convert the number 65_{10} to binary



Converting binary to decimal

Bit position

7	6	5	4	3	2	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Decimal value

Converting binary to decimal

Bit position

7	6	5	4	3	2	1	0
128	**64**	**32**	**16**	**8**	**4**	**2**	**1**

Decimal value

UCSC

BIT

Converting binary to decimal

Example:

Convert the unsigned binary number **10011010** to decimal

1	0	0	1	1	0	1	0

Converting binary to decimal

7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Converting binary to decimal

7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0
128	64	32	16	8	4	2	1

Converting binary to decimal

7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0
128	64	32	**16**	**8**	4	**2**	1

128 + 16 + 8 + 2 = *154*

So, **10011010** in unsigned binary is **154** in decimal

Converting binary to decimal

Example:

Convert the decimal number **105** to unsigned binary

Converting binary to decimal

Q. Does 128 fit into **105**?

A. **No**

7	6	5	4	3	2	1	0
0							
128	64	32	16	8	4	2	1

Next, consider the difference: 105- 0*128 = 105

Converting binary to decimal

Q. Does 64 fit into **105**?

A. **Yes**

7	6	5	4	3	2	1	0
0	1						
128	64	32	16	8	4	2	1

Next, consider the difference: 105- 1*64 = 41

Converting binary to decimal

Q. Does 32 fit into **41**?

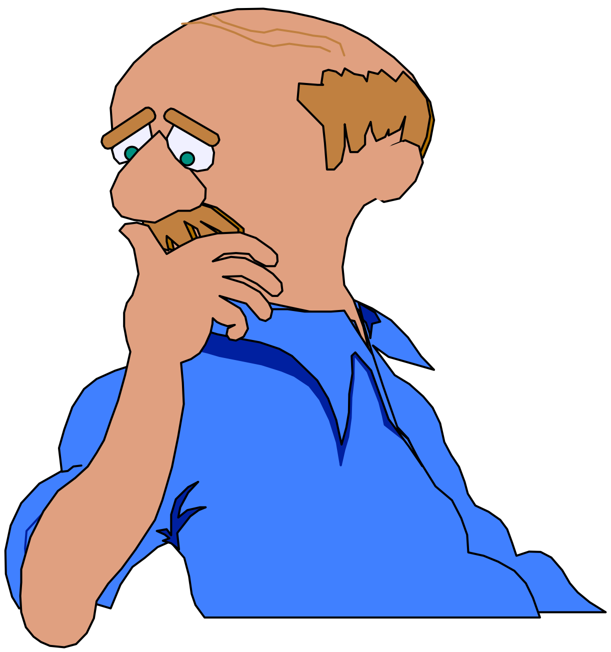
***A.* Yes**

7	6	5	4	3	2	1	0
0	1	1					
128	64	32	16	8	4	2	1

Next, consider the difference: 41- 32 = 9

Your turn

Convert the number 00110010_2 to decimal

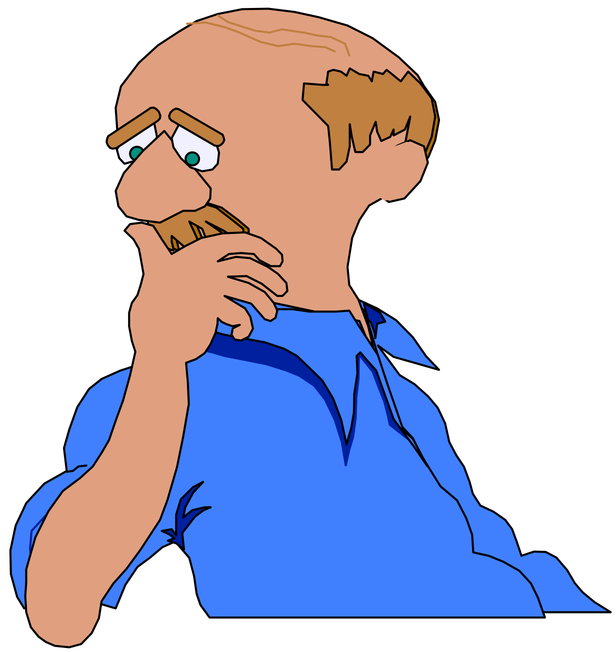


Converting binary numbers

- Decimal System
  - 0,1,2,3,4,5,6,7,8,9,10,11,12,13……
- Binary System
  - 0,1,10,11,100,101,110,111,1000,1001,1010,1011,1100,1101……

Your turn

Using 5 binary digits how many numbers you can represent?

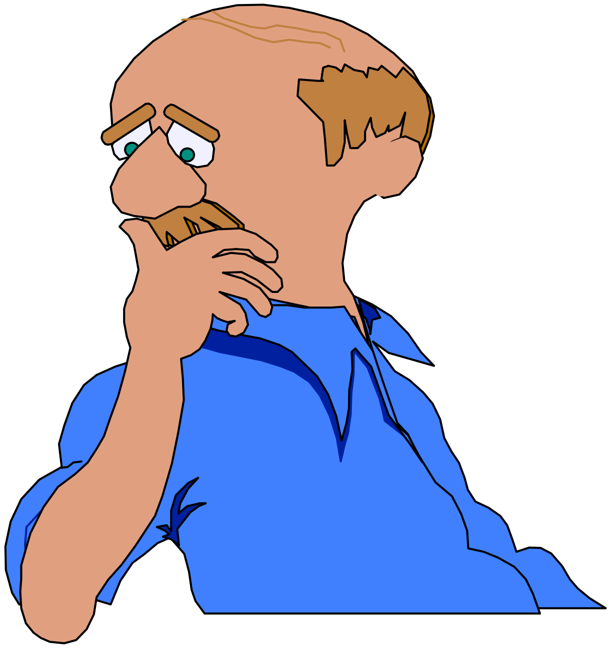


Hexadecimal Notation

HEX	Bit Pattern	HEX	Bit Pattern
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Your turn

- How many binary digits need to represent a hexadecimal digit?



Converting hexadecimal numbers

- Decimal System
  - 0,1,2,3,4,5,6,7,8,9,10,11,12,13……
- Hexadecimal System
  - 0,1,,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,13,14,15, 16,17,18,19,1A,1B,1C,1D,1E,1F…...

Binary to Hexadecimal Conversion

- 10010110_2
- **1001** **0110**
- **1001** **0110**
- **9** **6**

10010110_2 = **96** Hexadecimal

Binary to Hexadecimal Conversion

- 11011011_2
- **1101** **1011**
- **1101** **1011**
- **D** **B**

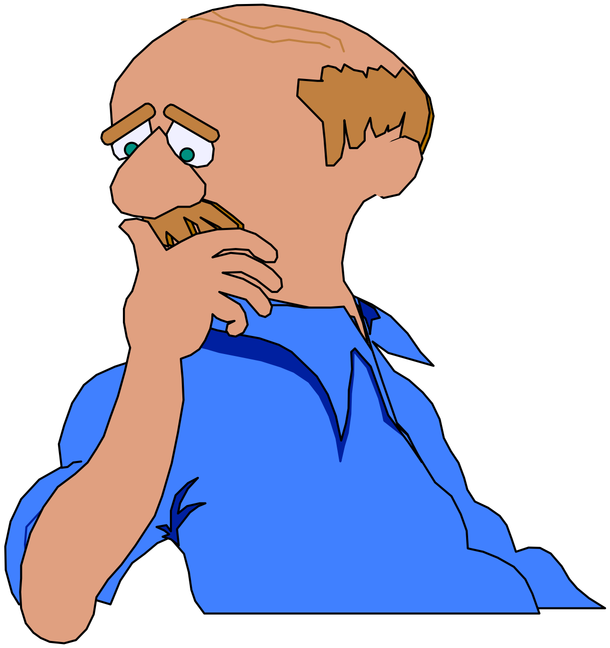
11011011_2 = **DB** Hexadecimal

Your turn

Convert the following binary string to Hexadecimal ...

00101001

11110101



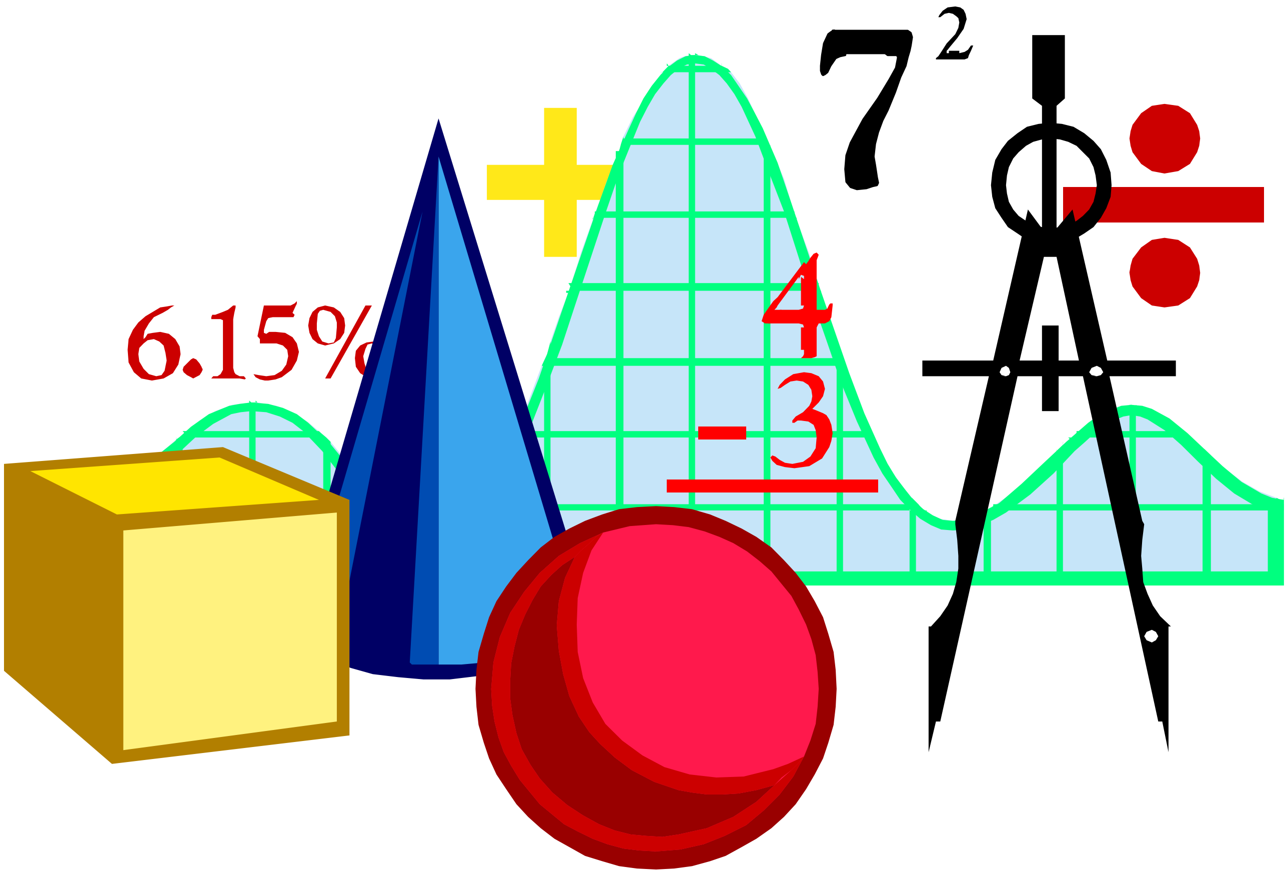
Binary to Hexadecimal Conversion

- $00101001\ 1110101_2$
- **00101001 11110101**
- **0010 1001 1111 0101**

  2 9 F 5

$00101001\ 1110101_2$ = **29F5** Hex

Computer Number System



ASCII Codes

- American Standard Code for Information Interchange (ASCII)

- Use bit patterns of length seven to represent
  - Letters of English alphabet: **a - z** and **A - Z**
  - Digits: **0 – 9**
  - Punctuation symbols: **(,), [,], {, }, ’, ”, !, /, **
  - Arithmetic Operation symbols: **+, -, *, <, >, =**
  - Special symbols: **(space), %, $, #, &, @, ^**

- $\mathbf{2^7 = 128}$ characters can be represented by ASCII

Character Representation: ASCII Table

Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII
(space)	00100000	A	01000001	a	01100001	0	00110000
!	00100001	B	01000010	b	01100010	1	00110001
“	00100010	C	01000011	c	01100011	2	00110010
#	00100011	D	01000100	d	01100100	3	00110011
$	00100100	E	01000101	e	01100101	4	00110100
%	00100101	F	01000110	f	01100110	5	00110101
&	00100110	G	01000111	g	01100111	6	00110110
…….		……..		…….		…….	

Character Representation: ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Character Representation: ASCII Table

- As computers became more reliable the need for parity bit faded.
  - Computer manufacturers extended ASCII to provide more characters, e.g., international characters
  - Used ranges (2^7) 128 ↔ 255 (2^8 - 1)

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	│	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	169	_	185	╣	201	╔	217	┘	233	Θ	249	·
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	█	235	δ	251	√
139	ï	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	_
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	^{2}
141	ì	158	_	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	░	192	└	208	╨	224	α	240	≡		

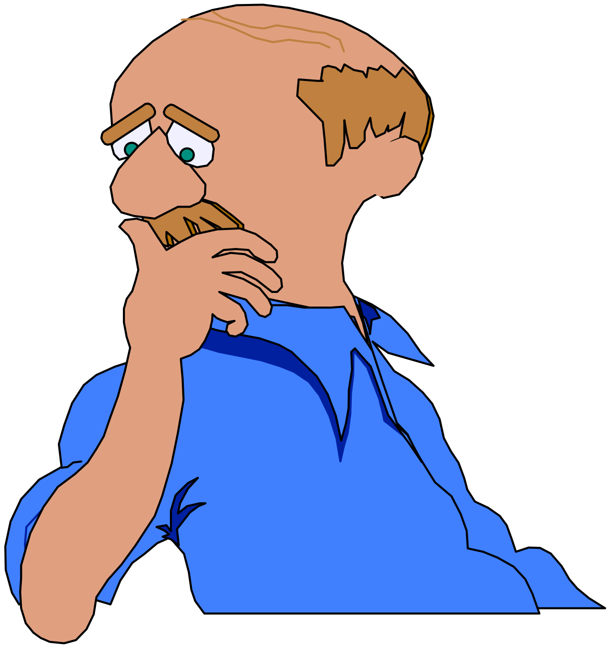
Source: www.LookupTables.com

UCSC

BIT

Your turn

- The BINARY string ...
- **0110101** can have two meanings!
- the CHARACTER **“5”** in ASCII
- AND ...
- the DECIMAL NUMBER **53** in BINARY Notation



Character Representation: Unicode

- EBCDIC and ASCII are built around the Latin alphabet
  - Are restricted in their ability for representing non-Latin alphabet
  - Countries developed their own codes for native languages
- Unicode: 16-bit system that can encode the characters of most languages
- 16 bits = 2^{16} = 65,636 characters

Character Representation: Unicode

- The Java programming language and some operating systems now use Unicode as their default character code
- Unicode codespace is divided into six parts
  - The first part is for Western alphabet codes, including English, Greek, and Russian
- Downward compatible with ASCII and Latin-1 character sets

Character Representation: Unicode

Character Types	Character Set Description	Number of Characters	Hexadecimal Values
Alphabets	Latin, Cyrillic, Greek, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Expansion or spillover from Han	4096	E000 to EFFF
User defined		4095	F000 to FFFE

UCSC

BIT

Character Representation: Example

- English section of Unicode Table
  - ➢ ACSII equivalent of A is 41_{16}
  - ➢ Unicode is equivalent of A:
    - $00\ 41_{16}$

	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074

- Full chart list:
  - ➢ http://www.unicode.org/charts/

Performing Arithmetic

Binary Addition

- 0 + 0 = **0**
- 0 + 1 = **1**
- 1 + 0 = **1**
- 1 + 1 = **10** (carry: **1**)

- E.g.

- **1 1 1 1 1** (carry)
  - 0 1 1 0 1
  - \+ 1 0 1 1 1
  - = **1 0 0 1 0 0**

Binary Subtraction

- 0 - 0 = **0**
- 0 - 1 = **1 (with borrow)**
- 1 - 0 = **1**
- 1 - 1 = **0**

• E.g.

- * * * (borrow)
  - 1 0 1 1 0 1
  - \- 0 1 0 1 1 1
  - = **0 1 0 1 1 0**

Binary Multiplication

- E.g.

$$
\begin{array}{r}
1\ 0\ 1\ 1 \\
\times\ 1\ 0\ 1\ 0 \\
\hline
0\ 0\ 0\ 0 \\
+\quad 1\ 0\ 1\ 1\ \\
+\quad 0\ 0\ 0\ 0\ \\
+\quad 1\ 0\ 1\ 1\ \\
\hline
= \mathbf{1\ 1\ 0\ 1\ 1\ 1\ 0} \\
\hline
\end{array}
$$

Binary Division

- E.g.

```
                 1 0 1
          ____________
  1 0 1  | 1 1 0 1 1
           - 1 0 1
           -------
               0 1 1
             - 0 0 0
             -------
                 1 1 1
               - 1 0 1
               -------
                   1 0
```

Representing Numbers

- Problems of number representation
  - Positive and negative
  - Radix point
  - Range of representation

- Different ways to represent numbers
  - Unsigned representation: non-negative integers
  - Signed representation: integers
  - Floating-point representation: fractions

Unsigned and Signed Numbers

- Unsigned binary numbers
  - Have **0** and **1** to represent numbers
  - Only positive numbers stored in binary
  - The Smallest binary number would be …

    0 0 0 0 0 0 0 0 which equals to **0**
  - The largest binary number would be …

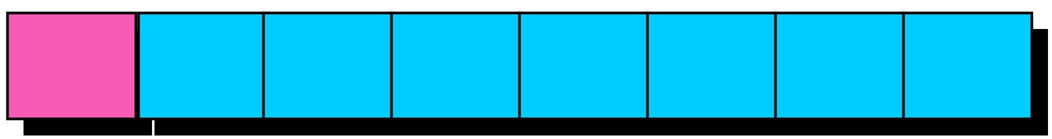
    1 1 1 1 1 1 1 1 which equals ….

    128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = **255** = $\mathbf{2^8-1}$
  - Therefore the **range** is **0 - 255** (**256** numbers)

Unsigned and Signed Numbers

- Signed binary numbers
  - Have **0** and **1** to represent numbers
  - The leftmost bit is a sign bit
    - **0** for positive
    - **1** for negative

Sign bit



Unsigned and Signed Numbers

- Signed binary numbers
  - The Smallest positive binary number is

    0 0 0 0 0 0 0 0 which equals to **0**

  - The largest positive binary number is

    0 1 1 1 1 1 1 1 which equals ….

    64 + 32 + 16 + 8 + 4 + 2 + 1 = **127 =** $\mathbf{2^7 - 1}$

  - Therefore the **range** for positive numbers is **0 - 127**
  - (**128** numbers)

Negative Numbers in Binary

- Problems with simple signed representation
  - Two representation of zero: **+ 0** and **– 0**
  - **0 0 0 0 0 0 0 0** and **1 0 0 0 0 0 0 0**
  - Need to consider both sign and magnitude in arithmetic
    - E.g. **5 – 3**
    - = **5 + (-3)**
    - = **0 0 0 0 0 1 0 1 + 1 0 0 0 0 0 1 1**
    - = **1 0 0 0 1 0 0 0**
    - = **-8**

Negative Numbers in Binary…

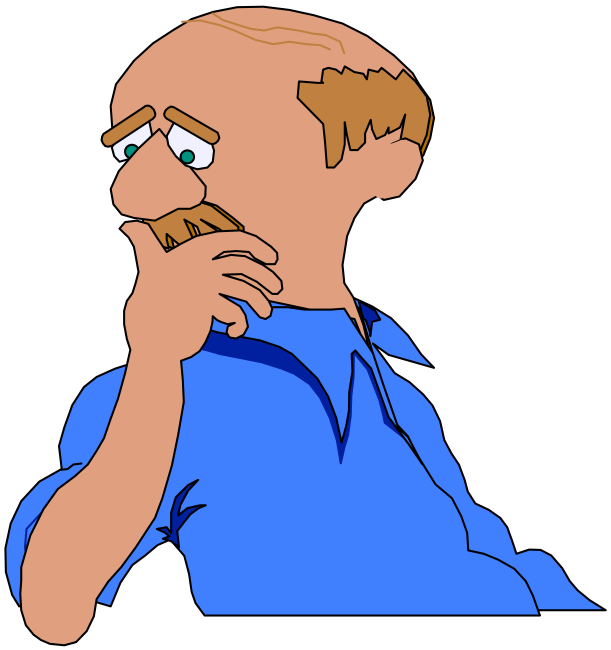
- Problems with simple signed representation
  - Need to consider both sign and magnitude in arithmetic
    - E.g. = **18 + (-18)**
    - = **0 0 0 1 0 0 1 0 + 1 0 0 1 0 0 1 0**
    - = **1 0 1 0 0 1 0 0**
    - = **-36**

Negative Numbers in Binary…

- The representation of a negative integer (**Two's Complement**) is established by:
  - Start from the **signed binary representation** of its **positive** value
  - Copy the bit pattern from **right** to **left** until a **1** has been copied
  - Complement the remaining bits: all the **1's** with **0's**, and all the **0's** with **1's**
  - **An exception: 1 0 0 0 0 0 0 0 = -128**

Your turn

What is the SMALLEST and LARGEST signed binary numbers that can be stored in 1 BYTE



Two's Compliment (8 bit pattern)

- 0 1 1 1 1 1 1 1 = +127
- .
- .
- 0 0 0 0 0 0 1 1 = +3
- 0 0 0 0 0 0 1 0 = +2
- 0 0 0 0 0 0 0 1 = +1
- 0 0 0 0 0 0 0 0 = 0

- 1 1 1 1 1 1 1 1 = -1
- 1 1 1 1 1 1 1 0 = -2
- 1 1 1 1 1 1 0 1 = -3
- .
- 1 0 0 0 0 0 0 1 = -127
- 1 0 0 0 0 0 0 0 = -128

UCSC

BIT

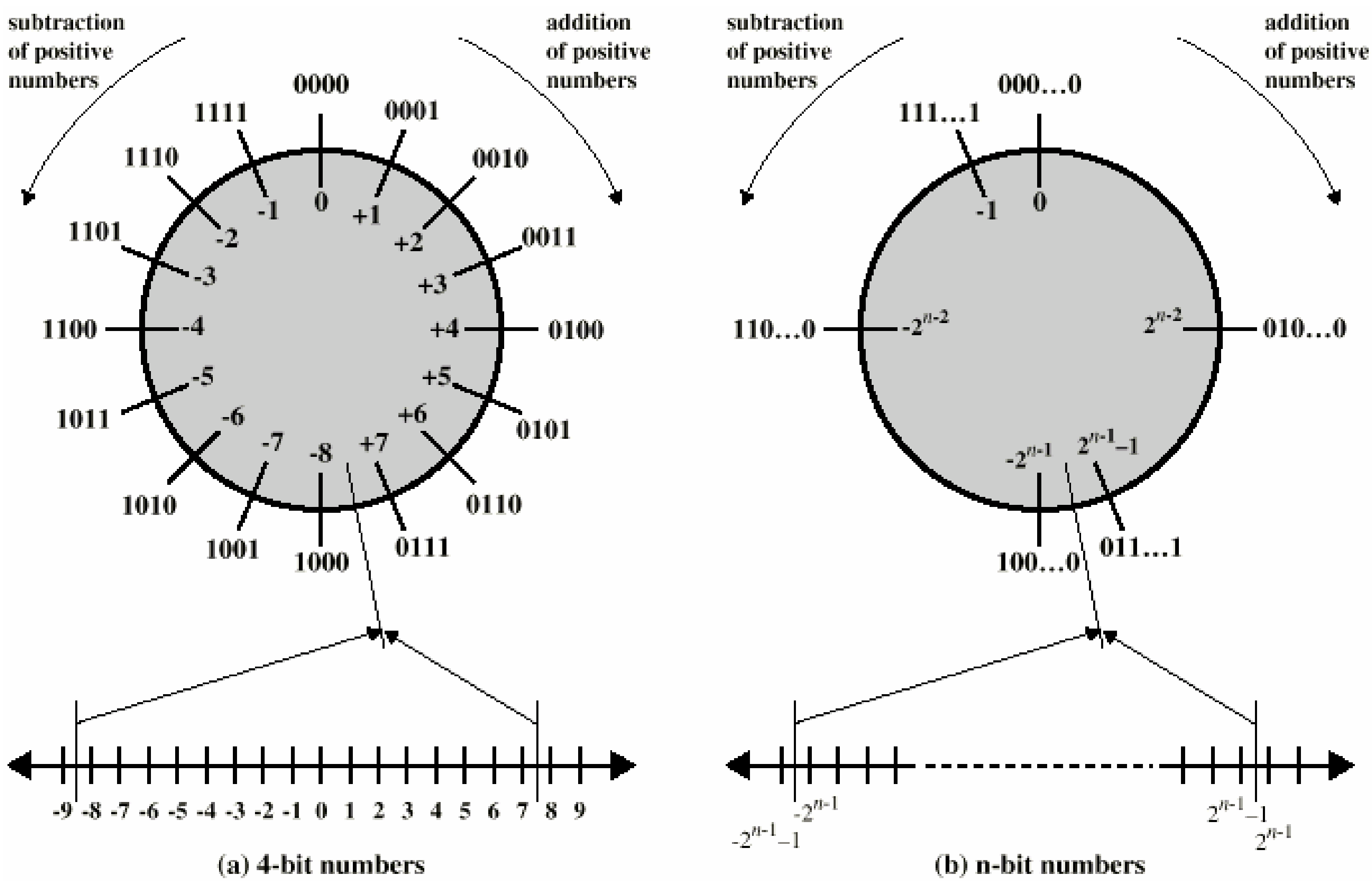
Two’s Compliment benefits

- One representation of zero
- Arithmetic works easily
- Negating is fairly easy

Ranges of Integer Representation

- 8-bit unsigned binary representation
  - Largest number: $\mathbf{1\ 1\ 1\ 1\ 1\ 1\ 1\ 1_2} = \mathbf{255_{10}}$
  - Smallest number: $\mathbf{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2} = \mathbf{0_{10}}$
- 8-bit two's complement representation
  - Largest number: $\mathbf{0\ 1\ 1\ 1\ 1\ 1\ 1\ 1_2} = \mathbf{127_{10}}$
  - Smallest number: $\mathbf{1\ 0\ 0\ 0\ 0\ 0\ 0\ 0_2} = \mathbf{-128_{10}}$
- The problem of **overflow**
  - $\mathbf{130_{10}} = \mathbf{1\ 0\ 0\ 0\ 0\ 0\ 1\ 0_2}$
  - $\mathbf{0\ 0\ 0\ 0\ 1\ 0_2}$ in two's complement

Geometric Depiction of Two's Complement Integers



Integer Data Types in C++

Type	Size in Bits	Range
unsigned int	16	0 – 65535
int	16	-32768 - 32767
unsigned long int	32	0 to 4,294,967,295
long int	32	-2,147,483,648 to 2,147,483,647

UCSC

BIT

Fractions in Decimal

- **16.357** = the SUM of ...

  $7 * 10^{-3} = {}^{7}/_{1000}$
  $5 * 10^{-2} = {}^{5}/_{100}$
  $3 * 10^{-1} = {}^{3}/_{10}$
  $6 * 10^{0} = 6$
  $1 * 10^{1} = 10$

- ${}^{7}/_{1000} + {}^{5}/_{100} + {}^{3}/_{10} + 6 + 10 = \mathbf{16\ {}^{357}/_{1000}}$

Fractions in Binary

- **10.011** = the SUM of ...

  $1 * 2^{-3} = {}^{1}/_{8}$

  $1 * 2^{-2} = {}^{1}/_{4}$

  $0 * 2^{-1} = 0$

  $0 * 2^{0} = 0$

  $1 * 2^{1} = 2$

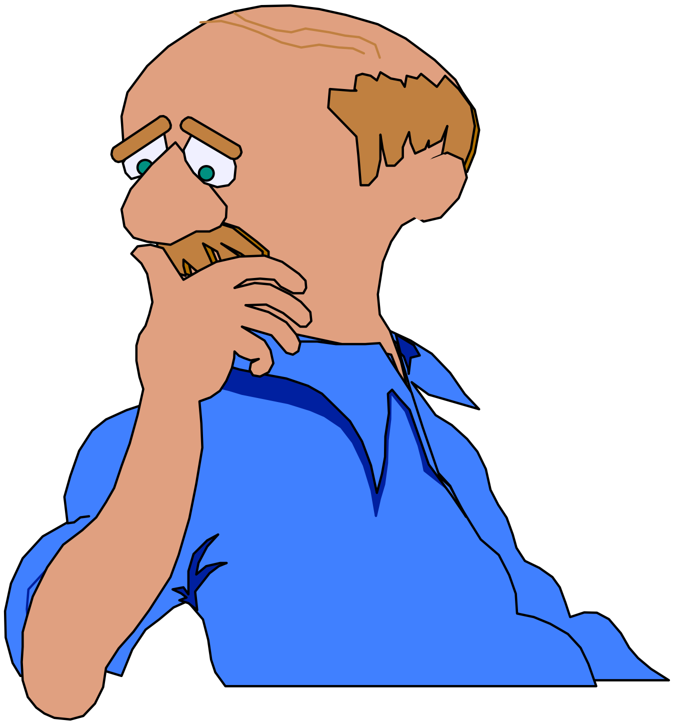
- ${}^{1}/_{8} + {}^{1}/_{4} + 2 =$ **$2\ {}^{3}/_{8}$**
- i.e. $10.011 = 2\ {}^{3}/_{8}$ in Decimal (Base 10)

UCSC

BIT

Your turn

What is **011.0101** in Base 10?



Fractions in Binary

- **011.0101** = the SUM of ...

  $1 * 2^{-4} = {}^{1}/_{16}$
  $0 * 2^{-3} = 0$
  $1 * 2^{-2} = {}^{1}/_{4}$
  $0 * 2^{-1} = 0$
  $1 * 2^{0} = 1$
  $1 * 2^{1} = 2$
  $0 * 2^{2} = 0$

- ${}^{1}/_{16} + {}^{1}/_{4} + 1 + 2 = \mathbf{3\ {}^{5}/_{16}}$

Decimal Scientific Notation

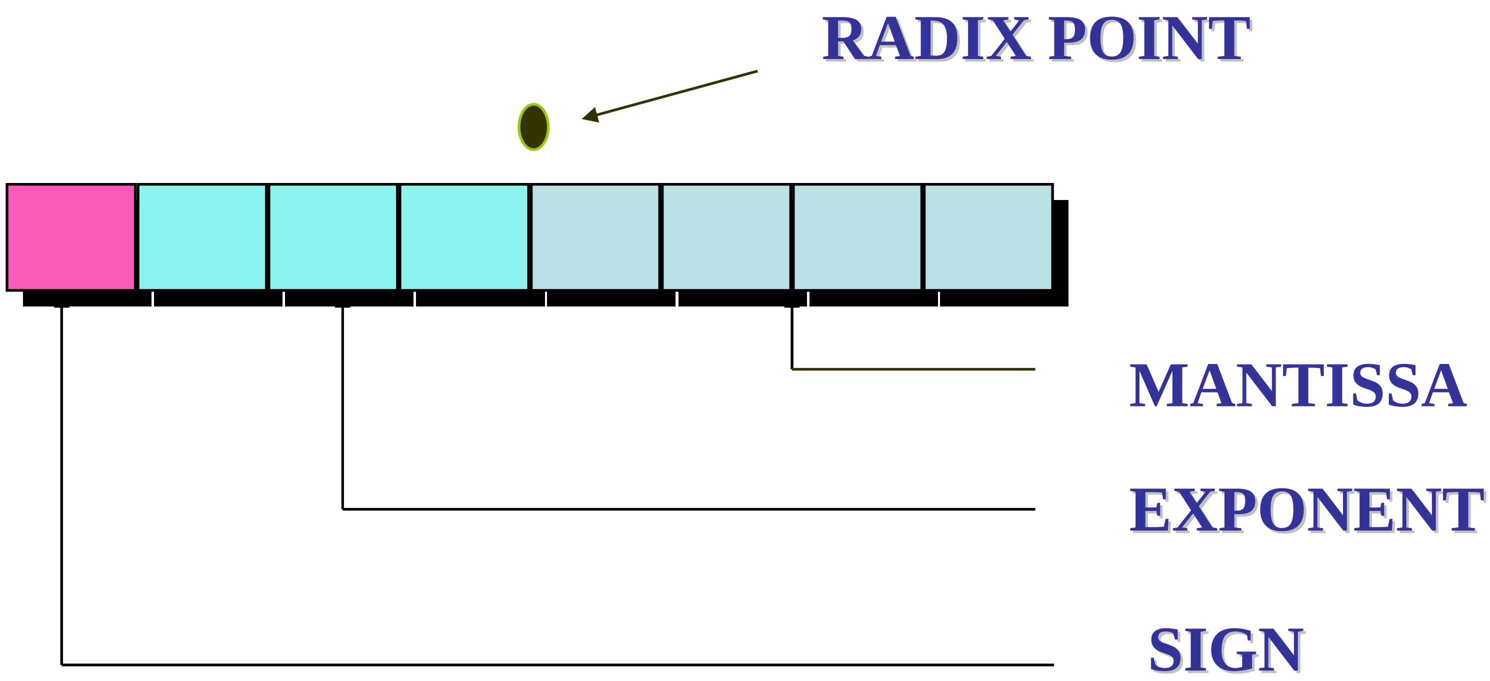
- Consider the following representation in decimal number …
  - **135.26** = **.13526** x 10^{3}
  - **13526000** = **.13526** x 10^{8}
  - **0.0000002452** = **.2452** x 10^{-6}

- **.13526** x 10^{3} has the following components:
  - a Mantissa = **.13526**
  - an Exponent = **3**
  - a Base = **10**

Floating Point Representation of Fractions

- Scientific notation for binary. Examples …
  - **11011.101 = 1.1011101** x 2^{4}
  - **-10110110000 = -1.011011** x 2^{10}
  - **0.00000010110111 = 1.0110111** x 2^{-7}

Floating Point Format in 1 Byte



SIGN = **0** (+ve) | **1** (-ve)
EXPONENT in **EXCESS FOUR** Notation

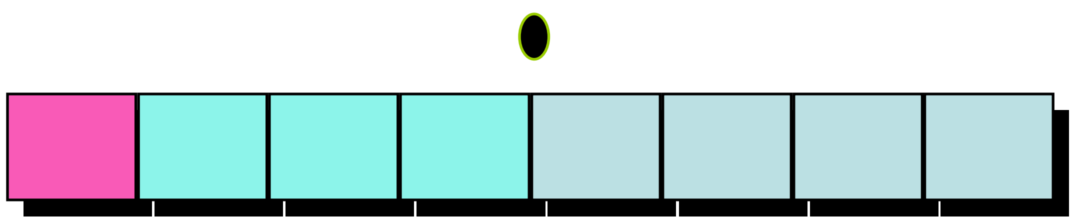
Floating Point Format in 1 Byte

•To STORE the number ...

$+1^1/_8 = 1.001$

in **FLOATING POINT NOTATION** ...

1. STORE the **SIGN BIT**



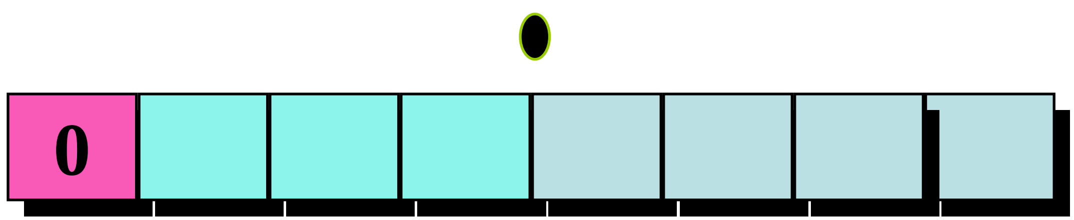
Floating Point Format in 1 Byte

•To STORE the number ...

$+1^{1}/_{8} = 1.001$

in **FLOATING POINT NOTATION** ...

1. STORE the **SIGN BIT**



UCSC

BIT

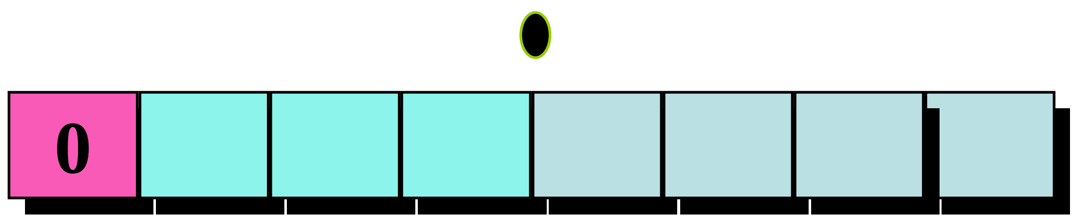
Floating Point Format in 1 Byte

•To STORE the number ...

$+1^1/_8 = 1.001$

in **FLOATING POINT NOTATION** ...

2. STORE the **MANTISSA BITS**



Floating Point Format in 1 Byte

•To STORE the number ...

$+1^{1}/_{8}$ = 1.001

in **FLOATING POINT NOTATION** ...

2. STORE the **MANTISSA BITS**

0				0	0	1	0

Floating Point Format in 1 Byte

- To STORE the number ...

  $$+1^{1}/_{8} = 1.001$$

in **FLOATING POINT NOTATION** ...

3. STORE the **EXPONENT BITS**

0				0	0	1	0

Excess-*k* Representation

Bit Pattern	Value Representation
111	4
110	3
101	2
100	1
011	0
010	-1
001	-2
000	-3

EXCESS THREE NOTATION

An excess notation system using bit pattern of length three

Excess-*k* Representation

- For N bit numbers, ***k*** is $2^{N-1}-1$
  - E.g., for 4-bit integers, ***k*** is **7**
- The actual value of each bit string is its
- unsigned value minus ***k***
- To represent a number in excess-***k***, add ***k***

Excess-*k* Representation

Unsigned **Excess-*k***

- [redacted]

0000....................

0001....................
0010....................
0011....................
0100....................
0101....................
0110....................
0111....................
1000....................
1001....................
1010....................
1011....................
1100....................
1101....................
1110....................
1111....................

[redacted]

0

$k = 7$

0

Floating Point Format in 1 Byte

•To STORE the number ...

$+1^{1}/_{8}$ = 1.001

in **FLOATING POINT NOTATION** ...

3. STORE the **EXPONENT BITS**

0	0	1	1	0	0	1	0

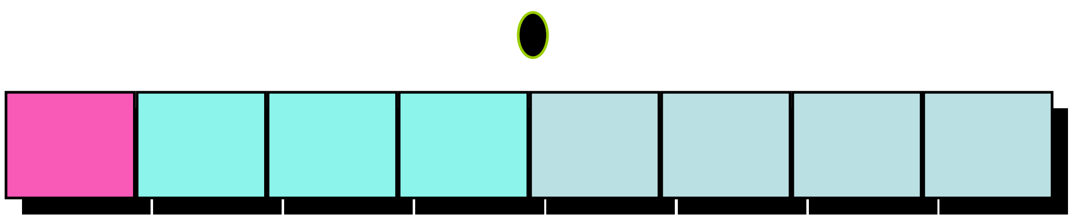
Floating Point Format in 1 Byte

•To STORE the number ...

$-3^{1}/_{4}$ = -11.01

in **FLOATING POINT NOTATION** ...

1. STORE the **SIGN BIT**



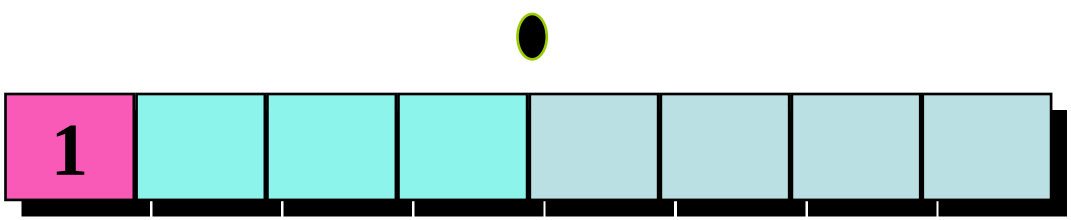
Floating Point Format in 1 Byte

•To STORE the number ...

$-3^{1}/_{4} = -11.01$

in **FLOATING POINT NOTATION** ...

1. STORE the **SIGN BIT**



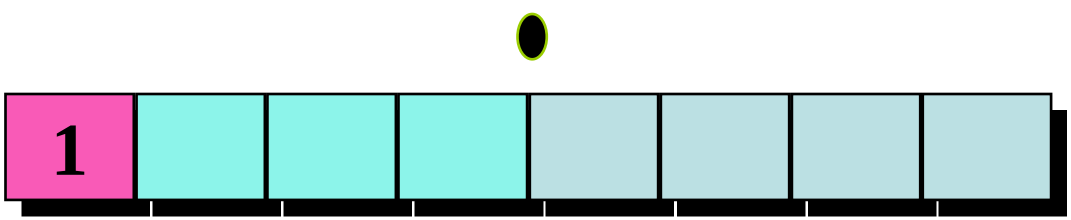
Floating Point Format in 1 Byte

•To STORE the number ...

$$-3^1/_4 = -11.01$$

in **FLOATING POINT NOTATION** ...

2. STORE the **MANTISSA BITS**



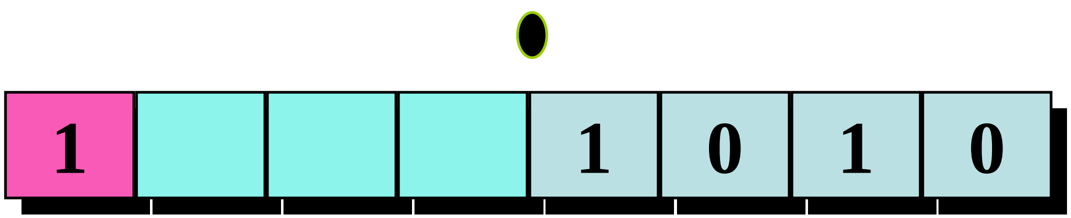
Floating Point Format in 1 Byte

- To STORE the number ...

  $-3^{1}/_{4} = -11.01$

in **FLOATING POINT NOTATION** ...

2. STORE the **MANTISSA BITS**



Floating Point Format in 1 Byte

•To STORE the number ...

$$-3^{1}/_{4} = -11.01$$

in **FLOATING POINT NOTATION** ...

3. STORE the **EXPONENT BITS**

1				1	1	0	1

Floating Point Format in 1 Byte

•To STORE the number ...

$-3^1/_4 = -11.01$

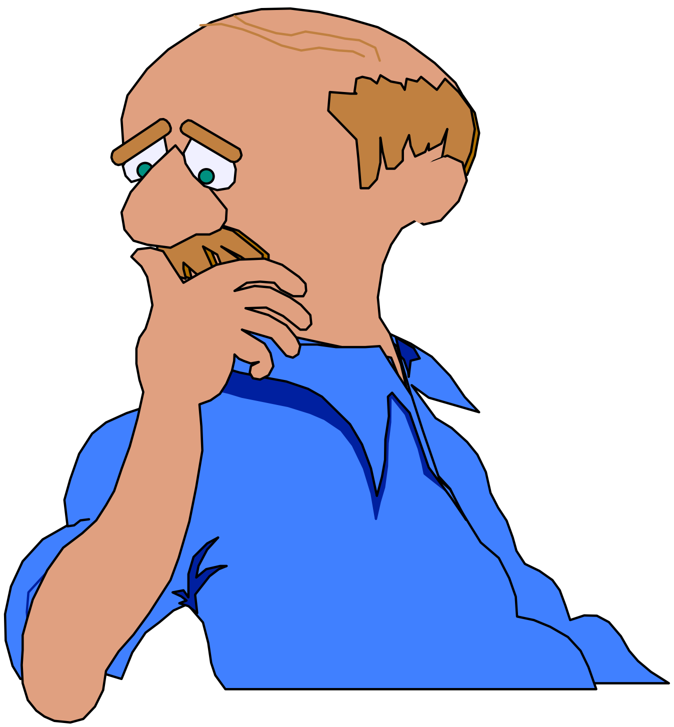
in **FLOATING POINT NOTATION** ...

3. STORE the **EXPONENT BITS**

1	1	0	0	1	1	0	1

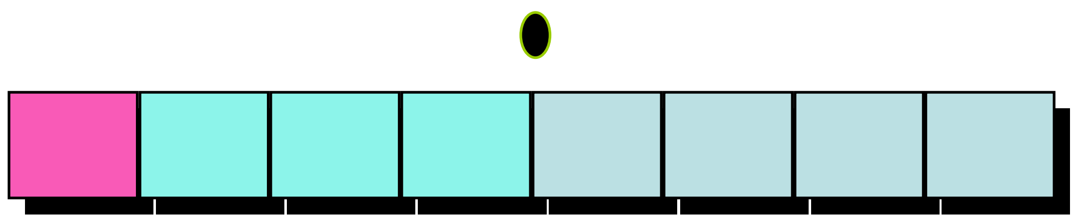
Your turn

Write down the **FLOATING POINT** form for the number $\mathbf{+^{11}/_{64}}$?



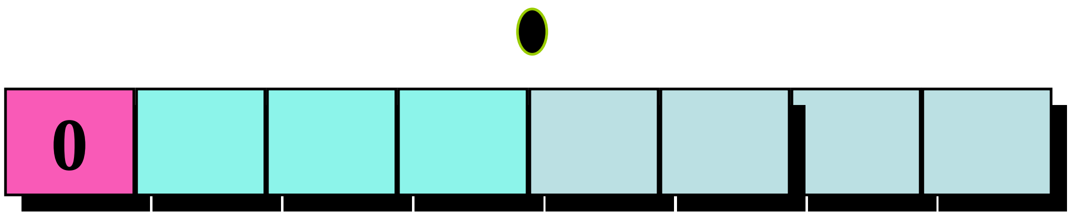
SOLUTION: $+{}^{11}/_{64}$ (.001011)

1. STORE the **SIGN BIT**



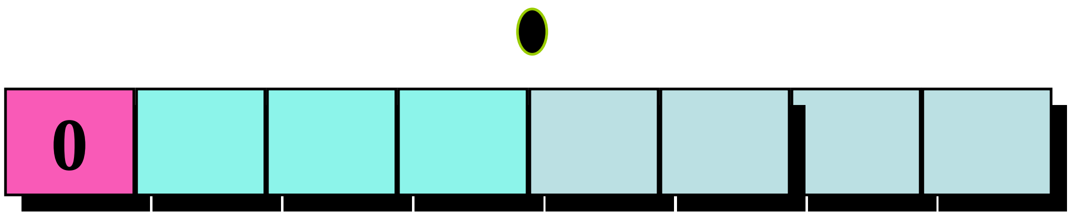
SOLUTION: $+^{11}/_{64}$ (.001011)

1. STORE the **SIGN BIT**



SOLUTION: $+{}^{11}/_{64}$ (.001011)

2. STORE the **MANTISSA BITS**



SOLUTION: $+{}^{11}/_{64}$ (.001011)

2. STORE the **MANTISSA BITS**

0				0	1	1	0

SOLUTION: $+^{11}/_{64}$ (.001011)

3. STORE the **EXPONENT BITS**

0	0	0	0	1	0	1	1

Converting FP Binary to Decimal

- Example ...
- **CONVERT 10111010 to decimal s**teps ...

1. Convert **EXPONENT** (EXCESS 4)
2. Apply **EXPONENT** to **MANTISSA**
3. Convert **BINARY Fraction**
4. Apply **SIGN**

SOLUTION: 10111010

1. CONVERT THE **EXPONENT**

1	0	1	1	1	0	1	0

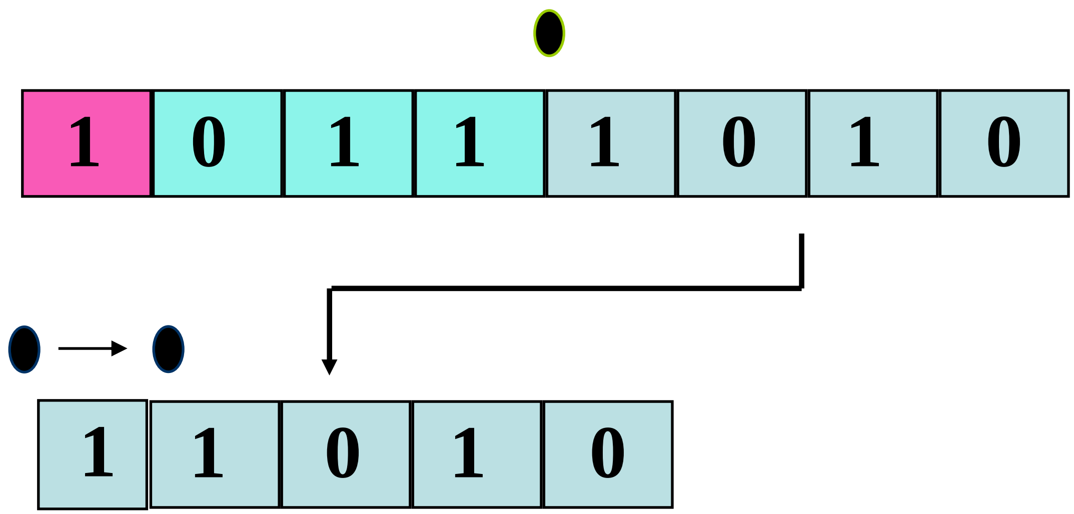
↓

0	1	1

→ $3 - 3 = 0$

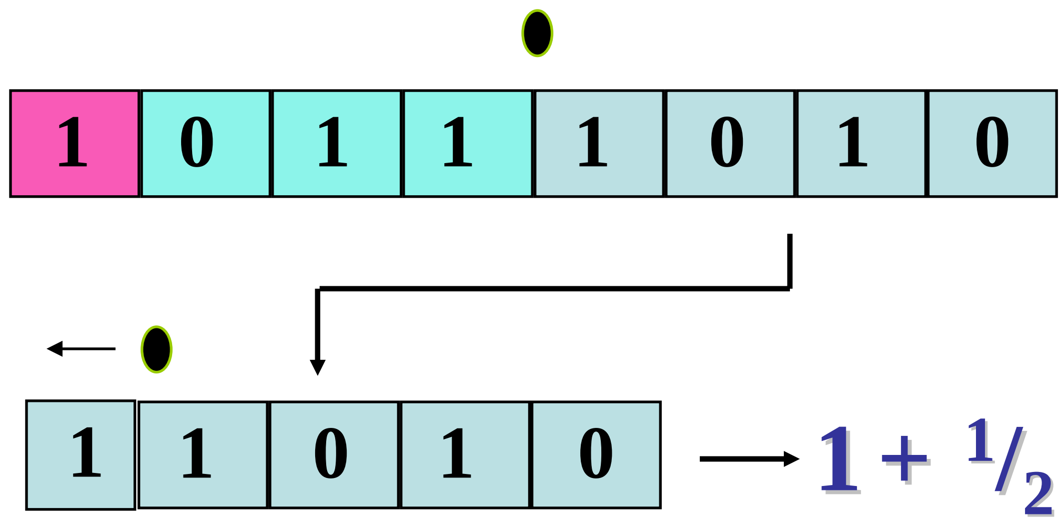
SOLUTION: 10111010

2. APPLY the **EXPONENT** to the **MANTISSA**



SOLUTION: 10111010

3. CONVERT from **BINARY FRACTION**



$$1 + {}^{1}/_{2} + {}^{1}/_{8} = 1\,{}^{5}/_{8}$$

SOLUTION: 10111010

4. APPLY the **SIGN**

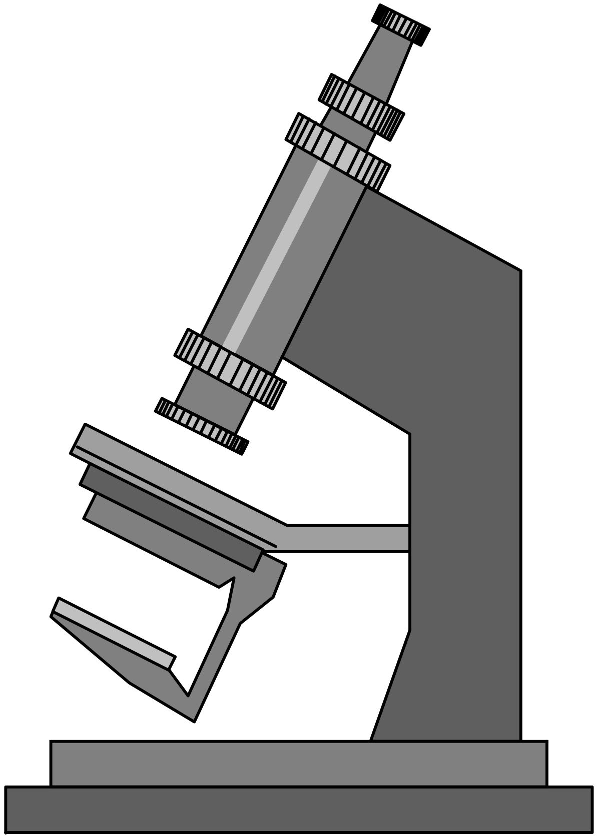
1	0	1	1	1	0	1	0

$$- \left(1 + {}^{1}/_{2} + {}^{1}/_{8} \right) = \quad -1\,{}^{5}/_{8}$$

ROUND-OFF ERRORS

•CONSIDER the FLOATING POINT Form of the number...

$+ 2\ ^{5}/_{16}$



ROUND-OFF ERRORS $+2^5/_8$

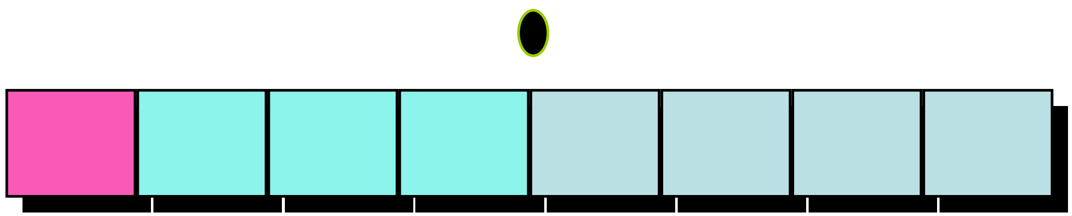
1. **CONVERT to BINARY FRACTION ...**

   $2^5/_8 = 10.0101$

   i.e. $2 + {}^1/_4 + {}^1/_{16}$

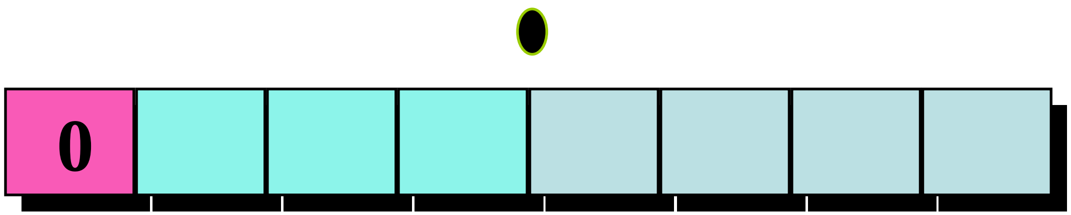
ROUND-OFF ERRORS $+2\,{}^{5}/_{8} = 10.101$

2. STORE THE **SIGN BIT** ...



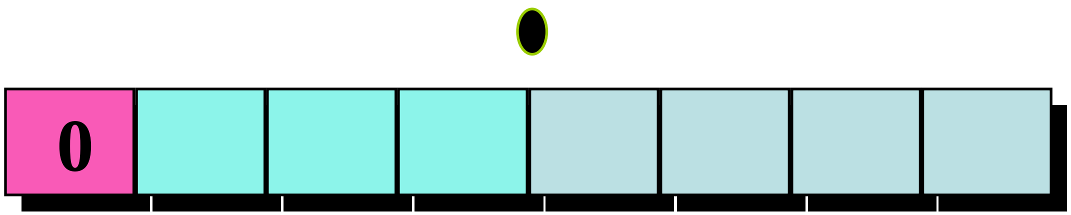
ROUND-OFF ERRORS $+2\,{}^{5}/_{8} = 10.101$

2. STORE THE **SIGN BIT** ...



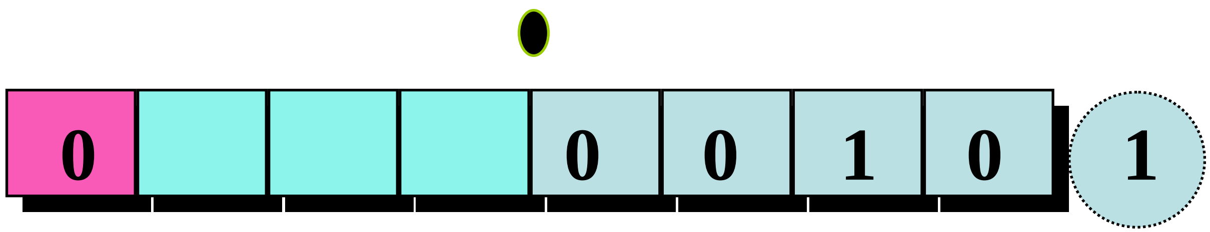
ROUND-OFF ERRORS $+2\,{}^{5}/_{8} = 10.101$

3. STORE THE **MANTISSA** ...



ROUND-OFF ERRORS $+2^{5}/_{8}$ = 10.101

3. STORE THE **MANTISSA** ...

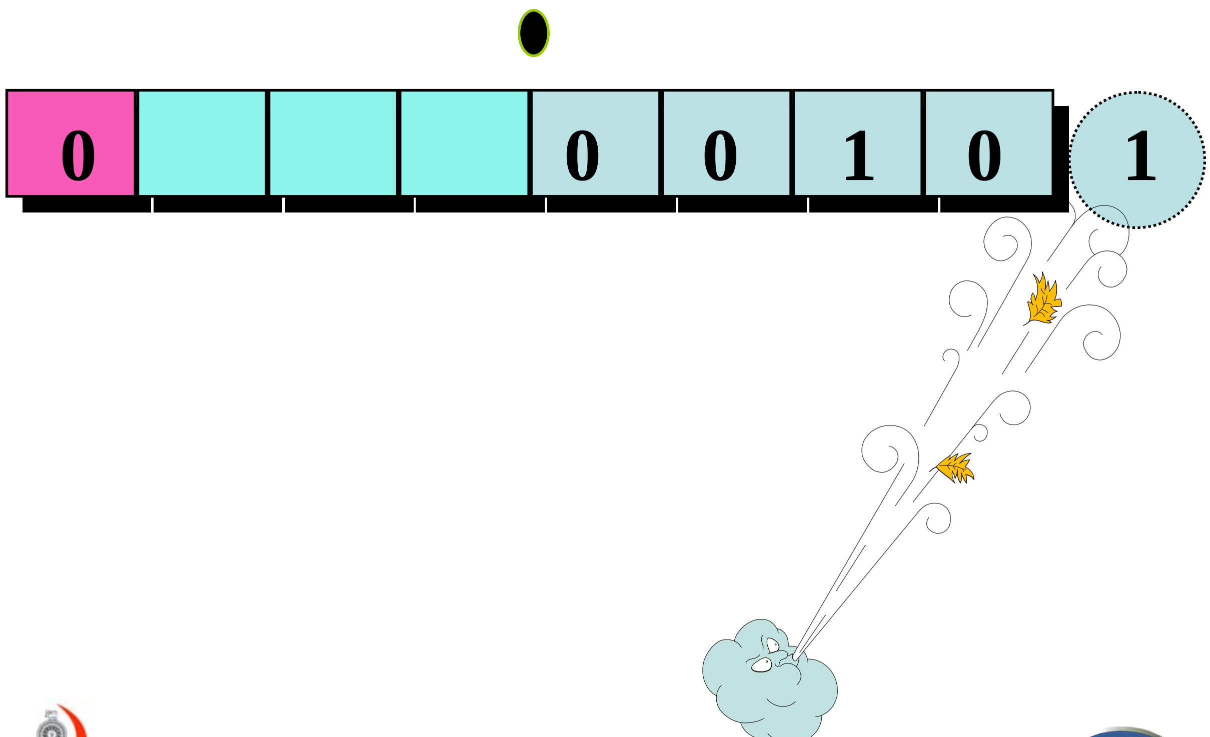


UCSC

BIT

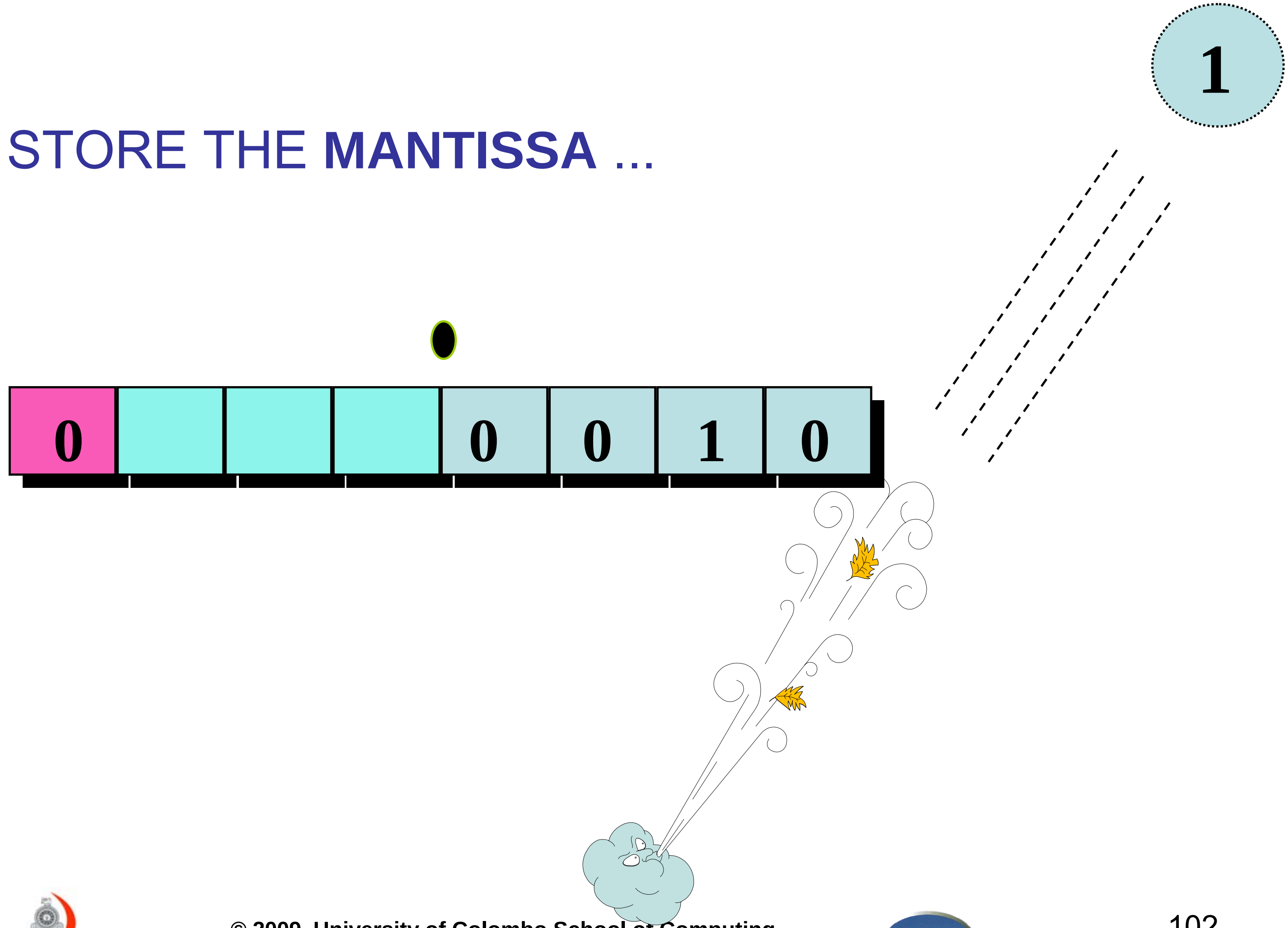
ROUND-OFF ERRORS $+2\,{}^{5}/_{8} = 10.101$

3. STORE THE **MANTISSA** ...



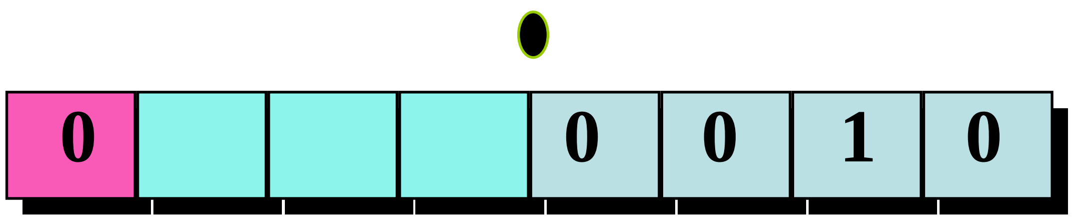
ROUND-OFF ERRORS $+2^5/_8 = 10.101$

3. STORE THE **MANTISSA** ...



ROUND-OFF ERRORS $+2\,{}^{5}/_{8} = 10.101$

4. STORE THE **EXPONENT** ...



ROUND-OFF ERRORS $+2\,{}^{5}/_{8} = 10.101$

4. STORE THE **EXPONENT** ...

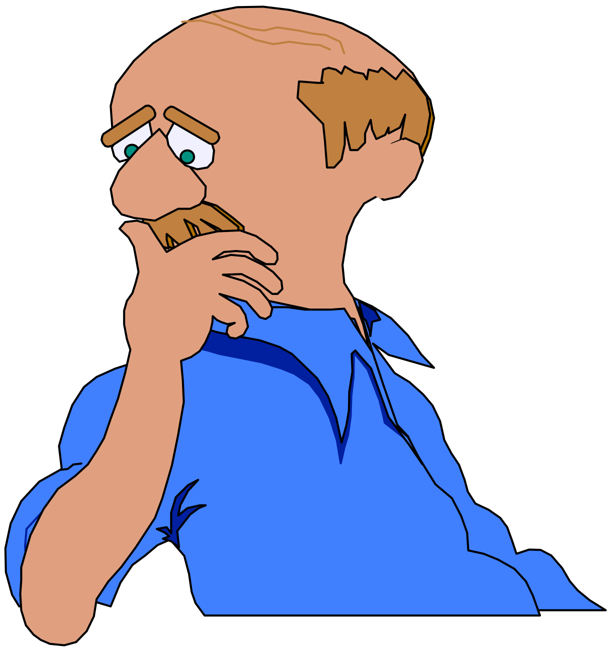
0	1	0	0	0	0	1	0

Converting this back to DECIMAL we get ...

$2\,{}^{1}/_{4}$ i.e. a ROUND OFF ERROR of ${}^{1}/_{16}$

Range of FP Representation

What is the BIGGEST and SMALLEST can be represented by one-byte floating point notation



Range of FP Representation

- The **biggest number** can be represented by one-byte floating point notation is:

0	1	1	1	1	1	1	1

- **$= +1.1111 \times 2^4 = +11111 = +31$**

Range of FP Representation

- The **Smallest positive number** can be represented by **one-byte floating point** notation is:

0	0	0	0	0	0	0	0

- **$= +1.0000 \times 2^{-3} = +.001 = +{}^{1}/_{8}$**

Range of FP Representation

- The **largest negative number** can be represented by **one-byte floating point** notation is:

1	0	0	0	0	0	0	0

- $= \mathbf{-1.0000 \times 2^{-3} \ = \ -.001 \ = \ -{}^{1}/_{8}}$

Range of FP Representation

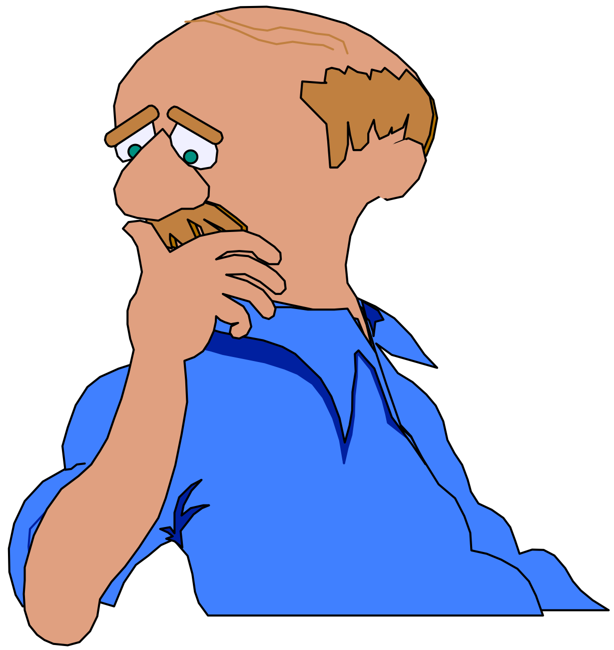
- The **smallest number** can be represented by **one-byte floating point** notation is:

1	1	1	1	1	1	1	1

- $= -1.1111 \times 2^4 = -11111 = -31$

Range of FP Representation

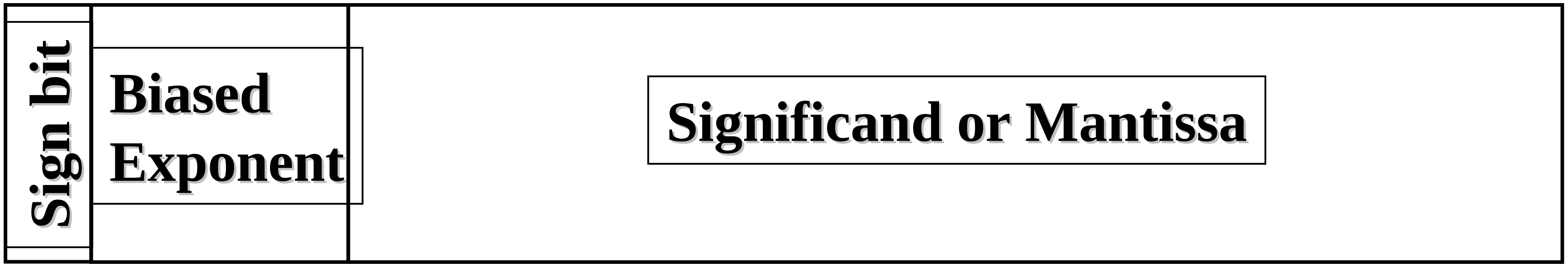
What is the SOLUTION for this???



Floating-Point Data types in C++

Type	Size in Bits	Range
float	32	3.4E-38 to 3.4E+38 Six digits of precision
double	64	1.7E-308 to 1.7E+308 Ten digits of precision
long double	80	3.4E-4932 to 3.4E+4932 Ten digits of precision

Floating-Point Representation



- **+/- . Mantissa** x 2 **^{exponent}**
- Point is actually fixed between sign bit and body of **Mantissa**
- **Exponent** indicates place value (point position)

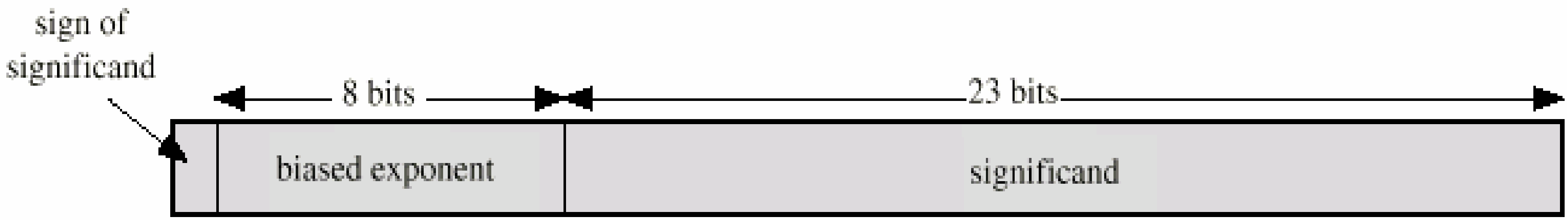
Floating-Point Representation

- **Mantissa** is stored in 2's compliment
- **Exponent** is in excess notation
  - 8 bit exponent field
  - Pure range is **0 – 255**
  - Subtract **127** to get correct value
  - Range **-127** to **+128**

Floating-Point Representation

- **Floating Point** numbers are usually normalized
- i.e. **exponent** is adjusted so that leading bit (MSB) of **mantissa** is 1
- Since it is always 1 there is no need to store it
- Where numbers are normalized to give a single digit before the decimal point
  - **E.g. 3.123×10^3**

Floating-Point Representation

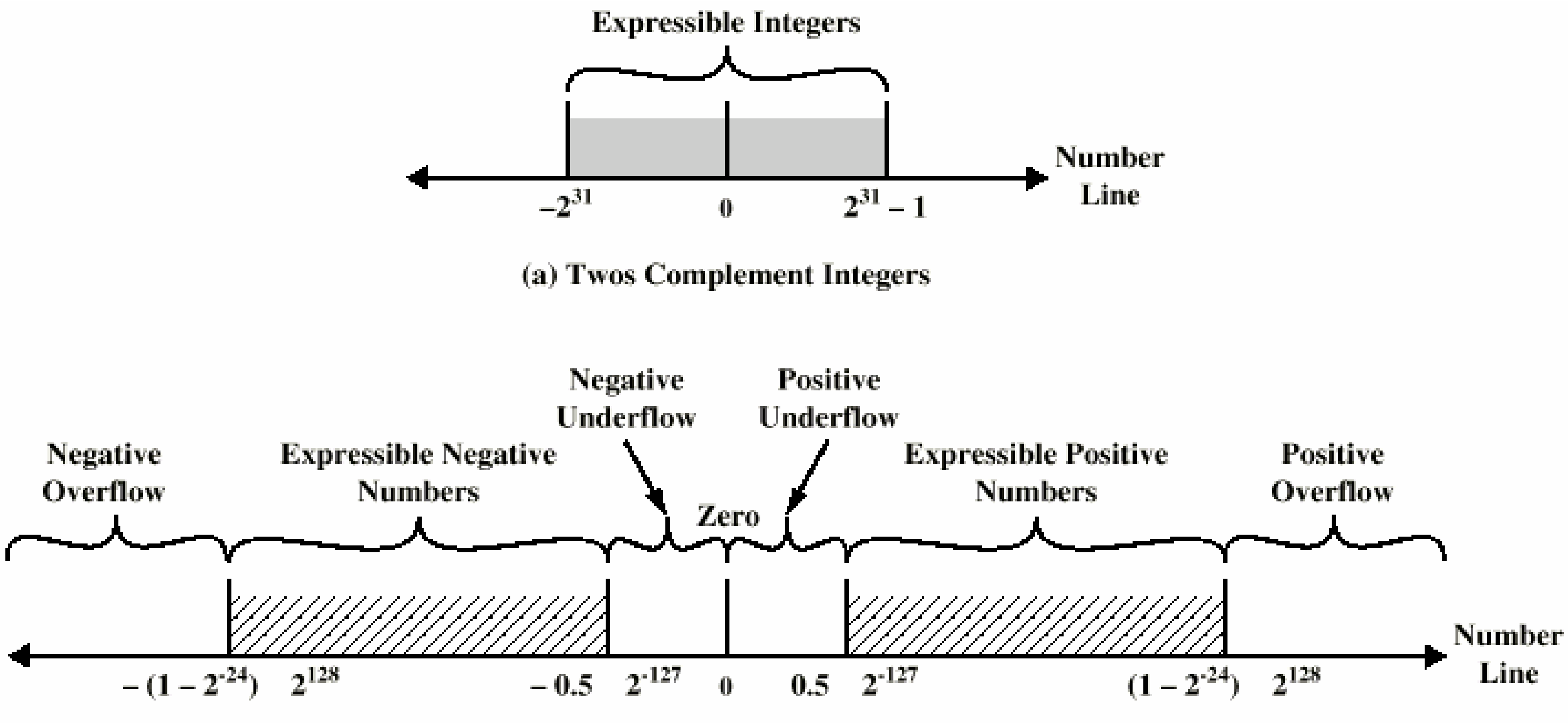


(a) Format

0.11010001	2^{10100}	=	0	10010011	10100010000000000000000
-0.11010001	2^{10100}	=	1	10010011	10100010000000000000000
0.11010001	2^{-10100}	=	0	01101011	10100010000000000000000
-0.11010001	2^{-10100}	=	1	01101011	10100010000000000000000

(b) Examples

Floating Point Representation: Expressible Numbers



(a) Twos Complement Integers

(b) Floating-Point Numbers

Representing the Mantissa

- The *mantissa* has to be in the range

  $1 \leq mantissa < base$

- Therefore
  - If we use base 2, the digit before the point must be a 1
  - So we don't have to worry about storing it
    ➔ We get 24 bits of precision using 23 bits
  - 24 bits of precision are equivalent to a little over 7 decimal digits:

$$\frac{24}{\log_2 10} \approx 7.2$$

Representing the Mantissa

- Suppose we want to represent π:
  3.14159265358979323846264338327 95.....

- That means that we can only represent it as:

  3.141592 (if we truncate)
  3.141593 (if we round)

Representing the Mantissa

- The IEEE standard restricts exponents to the range:

  $$-126 \leq exponent \leq +127$$

- The exponents –127 and +128 have special meanings:
  – If exponent = –127, the stored value is 0
  – If exponent = 128, the stored value is ∞

Floating Point Overflow

- Floating point representations can overflow, e.g.,

$$
\begin{array}{r}
1.111111 \times 2^{127} \\
+\ 1.111111 \times 2^{127} \\
\hline
11.111110 \times 2^{127}
\end{array}
$$

$$\boxed{1.1111110 \times 2^{128} = \infty}$$

Floating Point Underflow

- Floating point numbers can also get too *small*, e.g.,

$$
\begin{array}{r}
10.010000 \times 2^{-126} \\
\div\ 11.000000 \times 2^{0} \\
\hline
0.110000 \times 2^{-126}
\end{array}
$$

$$\boxed{1.100000 \times 2^{-127} = 0}$$

Floating Point Representation: Double Precision

IEEE-754 Double Precision Standard

- 64 bits:
  - 1 bit sign
  - 52 bit mantissa
  - 11 bit exponent
    - Exponent range is -1022 to +1023
    - $k = 2^{11-1}-1=1023$

Limitations

- Floating-point representations only approximate real numbers
- Using a greater number of bits in a representation can reduce errors but can never eliminate them
- Floating point errors
  - Overflow/underflow can cause programs to crash
  - Can lead to erroneous results / hard to detect

Floating Point Addition

Five steps to add two floating point numbers:

1. Express the numbers with the same exponent (*denormalize*)
2. Add the mantissas
3. Adjust the mantissa to one digit/bit before the point (*renormalize*)
4. Round or truncate to required precision
5. Check for overflow/underflow

Thank You